Developer tools

The below tools are text-based. UI developer tools are documented separately.

Workflow

- The listener
- Editor integration
- Runtime code reloading
- Unit testing
- Help system

Debugging

- The prettyprinter
- The inspector
- Stack effect tools
- Word annotations
- Deprecation tracking

Browsing

- Printing definitions
- Definition cross referencing
- Vocabulary hierarchy tools

Performance

- Timing code and collecting statistics
- Profiling code
- Object memory tools
- Listing threads
- Destructor tools
- Disassembling words

Deployment

- Application deployment

The listener

The listener evaluates Factor expressions read from the input stream. Typically, you write Factor code in a text editor, load it from the listener by calling **require**, **reload** or **run-file**, and then test it from interactively.

The classical first program can be run in the listener:

```
"Hello, world" print
Hello, world
```

New words can also be defined in the listener:

```
USE: math.functions
: twice ( word -- ) [ execute ] [ execute ] bi ; inline
81 \ sqrt twice .
3.0
```

Multi-line expressions are supported:

```
{ 1 2 3 } [
    .
] each
1
2
3
```

The listener will display the current contents of the datastack after every line of input.

The listener can watch dynamic variables:

Watching variables in the listener

Nested listeners can be useful for testing code in other dynamic scopes. For example, when doing database maintenance using the **db.tuples** vocabulary, it can be useful to start a listener with a database connection:

```
USING: db db.sqlite listener ;
"data.db" <sqlite-db> [ listener ] with-db
```

Starting a nested listener:

listener (--)

To exit a listener, invoke the **return** word.

The listener's mechanism for reading multi-line expressions from the input stream can be called from user code:

read-quot (-- quot/f)

Watching variables in the listener

The listener prints values of dynamic variables which are added to a watch list:

visible-vars

To add or remove a single variable:

show-var (var --)

hide-var (var --)

To add and remove multiple variables:

show-vars (seq --)

hide-vars (seq --)

Clearing the watch list:

hide-all-vars (--)

Editor integration

Factor development is best done with one of the supported editors; this allows you to quickly jump to definitions from the Factor environment.

edit (defspec --)

Depending on the editor you are using, you must load one of the child vocabularies of the **editors** vocabulary, for example **editors.emacs**:

```
USE: editors.emacs
```

If you intend to always use the same editor, it helps to have it load during stage 2 bootstrap. Place the code to load and possibly configure it in the **Bootstrap initialization file**.

Editor integration vocabularies store a quotation in a global variable when loaded:

edit-hook

If a syntax error was thrown while loading a source file, you can jump to the location of the error in your editor:

:edit (--)

Runtime code reloading

The **vocabs.refresh** vocabulary implements automatic reloading of changed source files.

With the help of the **io.monitors** vocabulary, loaded source files across all vocabulary roots are monitored for changes on disk.

If a change to a source file is detected, the next invocation of **refresh-all** will compare the file's checksum against its previous value, reloading the file if necessary. This takes advantage of the fact that the **source-files** vocabulary records CRC32 checksums of source files that have been parsed by **The parser**.

Words for reloading source files:

refresh (prefix --)

refresh-all (--)

Unit testing

A unit test is a piece of code which starts with known input values, then compares the output of a word with an expected output, where the expected output is defined by the word's contract.

For example, if you were developing a word for computing symbolic derivatives, your unit tests would apply the word to certain input functions, comparing the results against the correct values. While the passing of these tests would not guarantee the algorithm is correct, it would at least ensure that what used to work keeps working, in that as soon as something breaks due to a change in another part of your program, failing tests will let you know.

Unit tests for a vocabulary are placed in test files in the same directory as the vocabulary source file (see **Vocabulary loader**). Two possibilities are supported:

Tests can be placed in a file named `vocab -tests.factor`.
Tests can be placed in files in the `tests` subdirectory.

The latter is used for vocabularies with more extensive test suites.

If the test harness needs to define words, they should be placed in a vocabulary named `vocab.tests` where *vocab* is the vocab being tested.

Writing unit tests
Running unit tests

Writing unit tests

Assert that a quotation outputs a specific set of values:

unit-test

Assert that a quotation throws an error:

must-fail

must-fail-with

Assert that a quotation or word has a specific static stack effect (see **Stack effect checking**):

must-infer

must-infer-as

All of the above are used like ordinary words but are actually parsing words. This ensures that parse-time state, namely the line number, can be associated with the test in question, and reported in test failures.

Running unit tests

The following words run test harness files; any test failures are collected and printed at the end:

test (prefix --)

test-all (--)

The following word prints failures:

:test-failures (--)

Test failures are reported using the **Batch error reporting** mechanism and are shown in the **UI error list tool**.

Unit test failures are instances of a class, and are stored in a global variable:

test-failure

test-failures

Help system

The help system maintains documentation written in a simple markup language, along with cross-referencing and search. Documentation can either exist as free-standing *articles* or be associated with words.

Browsing documentation

Help topics are instances of a mixin:

topic

Most commonly, topics are article name strings, or words. You can display a specific help topic:

help (topic --)

You can also display the help for a vocabulary:

about (vocab --)

To list a vocabulary's words only:

words. (vocab --)

Examples

```
"evaluator" help

\ + help

"io.files" about
```

Writing documentation

By convention, documentation is written in files whose names end with `-docs.factor`. Vocabulary documentation should be placed in the same directory as the vocabulary source code; see **Vocabulary loader**.

A pair of parsing words are used to define free-standing articles and to associate documentation with words:

ARTICLE:

HELP:

A parsing word defines the main help article for a vocabulary:
ABOUT:

The *content* in both cases is a *markup element*, a recursive structure taking one of the following forms:
- a string,
- an array of markup elements,
- or an array of the form `{ $directive content... }`, where `$directive` is a markup word whose name starts with `$`, and `content...` is a series of markup elements

Here is a more formal schema for the help markup language:
```
<element> ::== <string> | <simple-element> | <fancy-element>
<simple-element> ::== { <element>* }
<fancy-element> ::== { <type> <element> }
```

Element types
Printing markup elements

Related words can be cross-referenced:
related-words (seq --)

See also
Help lint tool

Help lint tool
The **help.lint** vocabulary implements a tool to check documentation in an automated fashion. You should use this tool to check any documentation that you write.

To run help lint, use one of the following two words:
help-lint (prefix --)

help-lint-all (--)

Once a help lint run completes, failures can be listed:
:lint-failures (--)

Help lint failures are also shown in the **UI error list tool**.

Help lint performs the following checks:
- ensures examples run and produce stated output
- ensures **$see-also** elements don't contain duplicate entries
- ensures **$vocab-link** elements point to modules which actually exist
- ensures that **$values** match the stack effect declaration
- ensures that help topics actually render (this catches broken links, improper nesting, etc)

Tips of the day

The **help.tips** vocabulary provides a facility for displaying tips of the day in the **UI listener**.

Tips are defined with a parsing word:

TIP:

All tips defined so far:

All tips of the day

Help system implementation

Help topic protocol:

article-name (topic -- string)

article-title (topic -- string)

article-content (topic -- content)

article-parent (topic -- parent)

set-article-parent (parent topic --)

Boilerplate word help can be automatically generated (for example, slot accessor help):

word-help (word -- content)

word-help* (word -- content)

Help article implementation:

article (name -- article)

articles

Links:

link

>link (obj -- obj)

Utilities for traversing markup element trees:

elements (elt-type element -- seq)

collect-elements (element seq -- elements)

Links and **article** instances implement the definition protocol; refer to **Definitions**.

The prettyprinter

One of Factor's key features is the ability to print almost any object as a valid source literal expression. This greatly aids debugging and provides the building blocks for light-weight object serialization facilities.

Prettyprinter words are found in the **prettyprint** vocabulary.

The key words to print an object to **output-stream**; the first two emit a trailing newline, the second two do not:

. (obj --)

short. (obj --)

pprint (obj --)

pprint-short (obj --)

pprint-use (obj --)

The string representation of an object can be requested:

unparse (obj -- str)

unparse-use (obj -- str)

Utility for tabular output:

pprint-cell (obj --)

More prettyprinter usage:

Prettyprinting numbers
Prettyprinting stacks

Prettyprinter customization:

Prettyprint control variables
Extending the prettyprinter
Prettyprinter limitations

See also

Converting between numbers and strings, **Printing definitions**

Prettyprinting numbers

The **.** word prints numbers in decimal. A set of words in the **prettyprint** vocabulary is provided to print integers using another base.

.b (n --)

.o (n --)

.h (n --)

Prettyprinting stacks

Prettyprinting the current data, retain, call stacks:

.s (--)

.r (--)

.c (--)

Prettyprinting any stack:

stack. (seq --)

Prettyprinting any call stack:

callstack. (callstack --)

Note that calls to **.s** can also be included inside words as a debugging aid, however a more convenient way to achieve this is to use the annotation facility. See **Word annotations**.

Prettyprint control variables

The following variables affect the **.** and **pprint** words if set in the current dynamic scope:

tab-size

margin

nesting-limit

length-limit

line-limit

number-base

string-limit?

boa-tuples?

c-object-pointers?

The default limits are meant to strike a balance between readability, and not producing too much output when large structures are given. There are two combinators that override the defaults:

with-short-limits (quot --)

without-limits (quot --)

That the **short.** and **pprint-short** words wrap calls to **.** and **pprint** in **with-short-limits**. Code that uses the prettyprinter for serialization should use **without-limits** to avoid producing unreadable output.

Extending the prettyprinter

One can define literal syntax for a new class using the **The parser** together with corresponding prettyprinting methods which print instances of the class using this syntax.

Literal prettyprinting protocol
Prettyprinting more complex literals

The prettyprinter actually exposes a general source code output engine and is not limited to printing object structure.

Prettyprinter sections

Prettyprinter limitations

When using the prettyprinter as a serialization mechanism, keep the following points in mind:

When printing words, **USING:** declarations are only output if the **pprint-use** or **unparse-use** words are used.

Long output will be truncated if certain **Prettyprint control variables** are set.

Shared structure is not reflected in the printed output; if the output is parsed back in, fresh objects are created for all literal denotations.

Circular structure is not printed in a readable way. For example, try this:

```
{ f } dup dup set-first .
```

Floating point numbers might not equal themselves after being printed and read, since a decimal representation of a float is inexact.

On a final note, the **short.** and **pprint-short** words restrict the length and nesting of printed sequences, their output will very likely not be valid syntax. They are only intended for interactive use.

The inspector

The inspector displays a tabular view of an object and adds navigation and editing features. Inspector words are found in the **inspector** vocabulary.

Starting the inspector:

inspect (obj --)

The inspector supports a number of commands which operate on the most recently inspected object:

&push (-- obj)

&back (--)

&at (n --)

&put (value n --)

&add (value key --)

&rename (key n --)

&delete (n --)

A variable holding the current object:

me

A description of an object can be printed without starting the inspector:

describe (obj --)

Stack effect tools

Stack effect checking can be used interactively to print stack effects of quotations without running them. It can also be used from **Smart combinators**.

infer (quot -- effect)

infer. (quot --)

There are also some words for working with **effect** instances. Getting a word's declared stack effect:

stack-effect (word -- effect/f)

Converting a stack effect to a string form:

effect>string (obj -- str)

Comparing effects:

effect-height (effect -- n)

effect<= (effect1 effect2 -- ?)

effect= (effect1 effect2 -- ?)

The class of stack effects:

effect

effect? (object -- ?)

Word annotations

The word annotation feature modifies word definitions to add debugging code. You can restore the old definition by calling **reset** on the word in question.

Printing messages when a word is called or returns:

watch (word --)

watch-vars (word vars --)

Timing words:

reset-word-timing (--)

add-timing (word --)

word-timing. (--)

All of the above words are implemented using a single combinator which applies a quotation to a word definition to yield a new definition:

annotate (word quot --)

Warning

Certain internal words, such as words in the **math**, **sequences** and UI vocabularies, cannot be annotated, since the annotated code may end up recursively invoking the word in question. This may crash or hang Factor. It is safest to only define annotations on your own words.

Deprecation tracking

Factor's core syntax defines a **deprecated** word that can be applied to words to mark them as deprecated. Notes are collected and reported by the **Batch error reporting** mechanism when deprecated words are used to define other words.

deprecated

:deprecations (--)

Printing definitions

The **see** vocabulary implements support for printing out **Definitions** in the image.

Printing a definition:
see (defspec --)

Printing the methods defined on a generic word or class (see **Objects**):
see-methods (word --)

Definition specifiers implementing the **Definition protocol** should also implement the *see protocol* :
see* (defspec --)

synopsis* (defspec --)

Definition cross referencing

Definitions can answer a sequence of definitions they directly depend on:

uses (defspec -- seq)

An inverted index of the above:

get-crossref (-- crossref)

Words to access it:

usage (defspec -- seq)

smart-usage (defspec -- seq)

Tools for interactive use:

usage. (word --)

vocab-uses. (vocab --)

vocab-usage. (vocab --)

See also

Definitions, **Words**, **Printing definitions**

Vocabulary hierarchy tools

These tools operate on all vocabularies found in the current set of **vocab-roots**, loaded or not.

Loading vocabulary hierarchies:

load (prefix --)

load-all (--)

Getting all vocabularies from disk:

all-vocabs (-- assoc)

all-vocabs-recursive (-- assoc)

Getting all vocabularies from disk whose names which match a string prefix:

child-vocabs (prefix -- assoc)

child-vocabs-recursive (prefix -- assoc)

Words for modifying output:

no-roots (assoc -- seq)

no-prefixes (seq -- seq')

Getting **Vocabulary metadata** for all vocabularies from disk:

all-tags (-- seq)

all-authors (-- seq)

Timing code and collecting statistics

You can time the execution of a quotation in the listener:

time (quot --)

This word also collects statistics about method dispatch and garbage collection:

dispatch-stats. (--)

gc-events. (--)

gc-stats. (--)

gc-summary. (--)

A lower-level word puts timings on the stack, instead of printing:

benchmark (quot -- runtime)

You can also read the system clock directly; see **System interface**.

See also

Profiling code, **Word annotations**, **Calendar**

Profiling code

The **tools.profiler** vocabulary implements a simple call counting profiler.

Quotations can be passed to a combinator which calls them with the profiler enabled:

profile (quot --)

After a quotation has been profiled, call counts can be presented in various ways:

profile. (--)

vocab-profile. (vocab --)

usage-profile. (word --)

vocabs-profile. (--)

method-profile. (--)

Profiler limitations

See also

UI profiler tool, **Word annotations**, **Timing code and collecting statistics**

Profiler limitations

Certain optimizations performed by the compiler can inhibit accurate call counting:

Calls to open-coded intrinsics are not counted. Certain words are open-coded as inline machine code, and in some cases optimized out altogether; this includes stack shuffling operations, conditionals, and many object allocation operations.

Calls to **inline** words are not counted.

Calls to methods which were inlined as a result of type inference are not counted.

Tail-recursive loops will only count the initial invocation of the word, not every tail call.

Object memory tools

You can print object heap status information:

room. (--)

heap-stats. (--)

heap-stats (-- counts sizes)

You can query memory status:

data-room (-- data-room)

code-room (-- code-room)

A combinator to get objects from the heap:

instances (quot -- seq)

You can check an object's the heap memory usage:

size (obj -- n)

The garbage collector can be invoked manually:

gc (--)

See also

Images

Listing threads

Printing a list of running threads:

threads. (--)

Destructor tools

The **tools.destructors** vocabulary provides words for tracking down resource leaks.

debug-leaks?

disposables. (--)

leaks (quot --)

See also

Deterministic resource disposal

Disassembling words

The **tools.disassembler** vocabulary provides support for disassembling compiled word definitions. It uses the `libudis86` library on x86-32 and x86-64, and `gdb` on PowerPC.

See also **compiler.tree.debugger** and **compiler.cfg.debugger**.

disassemble (obj --)

Application deployment

The stand-alone application deployment tool, implemented in the **tools.deploy** vocabulary, compiles a vocabulary down to a native executable which runs the vocabulary's **MAIN:** hook. Deployed executables do not depend on Factor being installed, and do not expose any source code, and thus are suitable for delivering commercial end-user applications.

Most of the time, the words in the **tools.deploy** vocabulary should not be used directly; instead, use **Application deployment UI tool**.

You must explicitly specify major subsystems which are required, as well as the level of reflection support needed. This is done by modifying the deployment configuration prior to deployment.

Preparing to deploy an application

In order to deploy an application as a stand-alone image, the application's vocabulary must first be given a **MAIN:** hook. Then, a *deployment configuration* must be created.

Deploy tool usage

Once the necessary deployment flags have been set, the application can be deployed:

deploy (vocab --)

deploy-image-only (vocab image --)

For example, you can deploy the **hello-ui** demo which comes with Factor. Note that this demo already has a deployment configuration, so nothing needs to be configured:

```
"hello-ui" deploy
```

On Mac OS X, this yields a program named `Hello world.app`.
On Windows, it yields a directory named `Hello world` containing a program named `hello-ui.exe`.
On Unix-like systems (Linux, BSD, Solaris, etc), it yields a directory named `Hello world` containing a program named `hello-ui`.

On all platforms, running the program will display a window with a message.

Deploy tool implementation

The deployment tool works by bootstrapping a fresh image, loading the vocabulary into this image, then applying various heuristics to strip the image down to minimal size.

The deploy tool generates *staging images* containing major subsystems, and uses the staging images to derive the final application image. The first time an application is deployed using a major subsystem, such as the UI, a new staging image is made, which can take a few minutes. Subsequent deployments of applications using this subsystem will be much faster.

Deploy tool caveats

Behavior of **boa**

In deployed applications, the **boa** word does not verify that the parameters on the stack satisfy the tuple's slot declarations, if any. This reduces deploy image size but can make bugs harder to track down. Make sure your program is fully debugged before deployment.

Behavior of **execute(**

Similarly, the **execute(** word does not check word stack effects in deployed applications, since stack effects are stripped out, and so it behaves exactly like **execute-effect-unsafe**.

Behavior of **call-next-method**

The **call-next-method** word does not check if the input is of the right type in deployed applications.

Error reporting

If the **deploy-reflection** level in the configuration is low enough, the debugger is stripped out, and error messages can be rather cryptic. Increase the reflection level to get readable error messages.

Choosing the right deploy flags

Finding the correct deploy flags is a trial and error process; you must find a tradeoff between deployed image size and correctness. If your program uses dynamic language features, you may need to elect to strip out fewer subsystems in order to have full functionality.